Lab2 Report

物理内存管理

根据教程的提示可以完成 `merge_chunk` 和 `buddy_free_pages`。对于 `split_chunk` 和 `buddy_get_pages`，观察后者的传入参数，猜测我们需要根据传入的阶数大小分配一个指定大小的页，而分配页的工作由前者来完成。

根据 `split_chunk` 中的提示，对于一个传入的页，我们需要递归的将其拆分，对于拆分后得到得后一半 `chunk`，将其存入对应阶数的 `free_list` 中，对于前一半 `chunk` 若等于要求阶数则直接返回，大于则递归的对齐调用 `split_chunk` 函数。

```
if (chunk->order == order) {
            return chunk;
    }

    // find buddy chunk
    chunk->order -= 1;
    struct page *buddy_chunk = get_buddy_chunk(pool, chunk);
    if (buddy_chunk != NULL && buddy_chunk->allocated == 0) {
            buddy_chunk->order = chunk->order;
            // add to the free list
            list_add(&buddy_chunk->node, &(pool->free_lists[buddy_chunk-
>order].free_list));
            pool->free_lists[buddy_chunk->order].nr_free += 1;
    }

return split_chunk(pool, order, chunk);
```

对于 sLAB 分配器和 Kmalloc 相关的内容，根据注释提示即可完成，只需要搞明白相关数据结构的定义和使用方式即可，在此不多赘述。

页表管理

全部的函数都可以仿照教程给出的 `map_range_in_pgtbl_common` 来完成，需要更改的内容只有对页表项进行的具体操作而已，逐级查询页表完全是通用的。

思考题 1
如果要在操作系统中支持写时拷贝，需要配置页表描述符的哪些字段，并在页错误时如何处理?

当多个进程同时请求一个相同的数据时，若他们并不对该数据进行修改而只是读取，操作系统会让他们访问同一个地址而不是创建多份备份，只有当某个进程需要修改该数据时，操作系统才会创建一个拷贝给该进程，而其他进程见到的数据都还是原始的数据。

因为多个进程共用一份数据，需要保证他们都不能对该数据进行修改，因此需要给页表描述符中添加一个只读项。当触发页异常时，需检测是否属于写时拷贝的情况，若属于，操作系统需要分配一个新的物理页给触发异常的进程，并将其试图访问的地址的数据拷贝到这个物理页中。

思考题 2

实验 1 中没有为内核页表使用细粒度的映射，而是直接沿用了启动时的粗粒度页表，这会产生什么问题?

内存浪费，粗粒度的映射会将大块物理内存映射到页表中，即使该大块物理内存中只有一小部分是有效数据，其余的空间会被浪费。

缺页管理

根据代码提示完成即可，需要注意的是引入的新数据结构 `rb_tree`，其实与前面用到的链表的使用方式差不多。